

Active Coordination in Ad Hoc Networks

Christine Julien and Gruia-Catalin Roman

Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130
{julien, roman}@wustl.edu

Abstract. The increasing ubiquity of communicating mobile devices and vastly different mobile application needs have led to the emergence of middleware models for ad hoc networks that simplify application programming. One such system, EgoSpaces, addresses specific needs of individual applications, allowing them to define what data is included in their operating context using declarative specifications constraining properties of data, agents that own the data, hosts on which those agents are running, and attributes of the ad hoc network. In the resulting coordination model, application agents interact with a dynamically changing environment through a set of views, or custom defined projections of the set of data present in the surrounding ad hoc network. This paper builds on EgoSpaces by allowing agents to assign behaviors to their personalized views. Behaviors consist of actions that are automatically performed in response to specified changes in a view. Behaviors discussed in this paper encompass reactive programming, transparent data migration, automatic data duplication, and event capture. Formal semantic definitions and programming examples are given for each behavior.

1 Introduction

The mobile ad hoc environment is an extreme network where the lack of an infrastructure necessitates a reinvestigation of communication paradigms. These opportunistically formed networks change rapidly in response to nodes entering and leaving communication range. Roving robots on an uninhabited planet may explore the terrain and coordinate to assimilate collected information. Automobiles on a highway communicate to gather traffic information. Rescue workers in a disaster recovery scenario must coordinate to perform their tasks quickly and safely, but the communication infrastructure is often crippled or destroyed. These domains demonstrate the potential for a wealth of applications requiring coordination among mobile components.

Much research focuses on developing protocols tailored to the specialized needs of these constrained networks. Ad hoc routing protocols [1–4] have made great strides to provide communication among groups of connected hosts, bringing the possibility of large scale ad hoc networks closer to reality. In such environments, the massive amounts of available information quickly overwhelms

applications, yet this information serves as the context for an application operating in the network, and applications need to adapt to changes in this context.

An application’s desire for adaptability manifests itself in the diversity of context-aware applications for traditional networks [5, 6]. FieldNote [7] allows researchers to implicitly attach context information to research notes, while tour guides [8, 9] display information based on the user’s location. The radically different properties of ad hoc networks, however, require new context-awareness models tailored to the environment’s specific complexities. The Context Toolkit [10] and Context Fabric [11] take steps to generalize context in various environments, but they do not address the need for a distributed coordination model.

Applications in this information-rich environment require coordination to manage, operate over, and react to context. As the demand for new applications grows, producing these applications places an increasingly heavy burden on programmers. Research in mobile computing middleware has shown that providing coordination constructs in middleware can simplify programming. While early middleware solutions focused on localizing reactions to individual hosts [12] or limiting interactions to symmetric communication [13], the EgoSpaces model [14] introduced asymmetric coordination, giving each application direct control over the size and scope of its personalized context. This approach is essential to accommodating programming for large, dense ad hoc networks.

Given the amount of context data in an ad hoc environment and the coordination constructs that have proven historically useful, the basic operations provided by EgoSpaces fall short of the abstractions required for rapid application development. This paper extends EgoSpaces to provide high-level coordination, including reactive programming, data migration, data duplication, and event capture. Of particular interest is our ability to reduce the specialized behaviors to a single construct, the reaction, giving promise for an efficient implementation that maximizes application responsiveness and minimizes communication overhead without sacrificing simplicity. The next section reviews EgoSpaces. Section 3 presents the extensions. Section 4 addresses performance considerations in presenting the constructs’ implementations. Conclusions appear in Section 5.

2 The EgoSpaces Coordination Model

EgoSpaces introduced an agent-centered context whose scope extends beyond the local host to contain data and resources associated with hosts and agents surrounding the agent of interest. This novel asymmetric coordination accommodates high-density and wide-coverage ad hoc networks.

2.1 Computational Model

EgoSpaces considers systems entailing both logically mobile agents (units of modularity and execution) operating over physically mobile hosts. Communication among agents and agent migration can occur whenever the hosts involved are connected. A closed set of these connected hosts defines an ad hoc network.

EgoSpaces bases coordination on a Global Virtual Data Structure (GVDS) [15], in which all data distributed among agents in the network appears, to the programmer, to be stored in a single, common data structure. At any given time, the available data depends on connectivity. Each agent maintains a local data repository, and, when agents move within communication range, their data structures logically merge to form a single “global” structure. The programmer interacts with the data via the standard operations for the data structure.

2.2 View Concept

In principle, an agent’s context includes all data available in the entire ad hoc network. EgoSpaces structures data access in terms of *views*, projections of the GVDS. Since one’s context is relative, the term *reference agent* denotes the agent whose context we are considering. Each agent defines individualized views by providing declarative specifications constraining properties of the network, hosts, agents, and data. As an example, imagine a building with a fixed infrastructure of sensors and information appliances providing context information. Sensors provide information regarding the building’s structural integrity, frequency of sounds, movement of occupants, etc. Engineers and inspectors carry PDAs that provide additional context data and assimilate information. Different people have specific tasks and therefore use information from different sensors. As an engineer moves through the building, he wishes to see structural information not for the whole building, but for the floors adjacent to his current floor. An agent running on his PDA might declare the following view:

Data from the past hour (reference to data) gathered by structural agents (reference to agents) on sensors (reference to hosts) within one floor of my current location (property of reference host).

Fig. 1 depicts this context, where the shaded area is the view of the engineer (in the hard hat). The inspector’s PDA does not currently fall within the context. Because EgoSpaces automatically maintains views, as the engineer moves, his view updates to include data from different sensors and devices.

In EgoSpaces, each agent specifies an individualized *access control function* that limits the ability of other agents to access its data. From the opposite direction, when an agent specifies a view, it attaches to the view a set of credentials that verify it to other agents. The specifying agent also declares the operations it intends to perform on the view. When determining the contexts of a view, EgoSpaces evaluates each contributing agent’s access control function over the

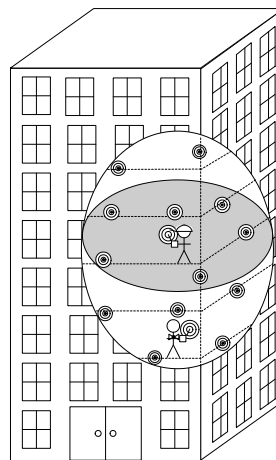


Fig. 1. Example view definition.

view’s credentials and operations. The access control function is evaluated for each individual data item, which provides a fine level of granularity. The view contains only data items that qualify via the access control function. More details on view specification and maintenance can be found in [14] and [16].

2.3 Basic Data Access Operations

In EgoSpaces, each agent carries its own local tuple space, and, when connectivity is available, connected agents’ tuple spaces merge into a GVDS. The operations provided over views are variations of the standard Linda [17] tuple space operations for tuple creation (**out**), tuple reading (**rd**), and tuple removal (**in**).

A tuple in EgoSpaces is a set of unordered triples of the form:

$$\langle (name, type, value), (name, type, value), \dots \rangle.$$

where the *names* of the fields must be unique within the tuple. The retrieval operations (**rd** and **in**) operate by matching a pattern against tuples in a view. Patterns constrain a tuple’s fields. To match a tuple, every constraint in the pattern must be satisfied by a corresponding field in the tuple.

Agents create tuples using **out** operations. Any tuple in an agent’s local tuple space is available in any view whose constraints it satisfies. To read and remove tuples, agents use variations of **rd** and **in** operations restricted to individual views. Because **in** operations remove tuples from the tuple space, they may affect other views if the tuple removed appears in multiple views. The **rd** and **in** operations block the issuing agent until a matching tuple exists and then return the match. If more than one tuple matches, one is chosen non-deterministically. Variations of these operations include blocking aggregate operations (**rdg** and **ing**) that return all matches, and probing versions of both single (**rdp** and **inp**) and aggregate operations (**rdgp** and **ingp**) which return ϵ if no match immediately exists. All operations listed thus far act over the view atomically, requiring a transaction over view participants. Because this can become costly, EgoSpaces offers scattered probe operations (**rdsp**, **insp**, **rdgsp**, and **ingsp**) that provide a weaker consistency because they do not lock the entire view and are allowed to miss a matching tuple. All operations and their semantics are provided in [14] with a formal description of tuples, patterns, and the matching function.

Programming Example. The building engineer might retrieve structural information about a single floor, perform local processing on the data, and then output a tuple indicating the current floor’s structural integrity. In this case, the engineer’s agent does not want to consume the data because it might be useful to other applications. The following code accomplishes this in EgoSpaces:

```

ν = [data from structural agents on the current floor]
p = ⟨⟨(strain, number, any), (acoustic emission, number, any),
    (time, time, [within 10 minutes])⟩⟩
data[] = ν.rdgp(p)
[local processing using data]
result = [tuple containing result]
out(result)

```

The first line creates a view; view specification details are omitted. In the definition of the pattern `p`, the constraint `any` indicates that the tuple must contain a field with the indicated name and type but the value is unrestricted.

3 Extending EgoSpaces

Many applications will require more sophisticated coordination mechanisms than those already presented. This section presents several additional constructs, including a powerful reactive mechanism, data migration, data duplication, and event capture. We show how using these sophisticated constructs eases the programming task and increases code encapsulation and reusability.

3.1 Advanced Constructs

Using the previous constructs, if an agent needs to wait for a piece of data before continuing, it must either block or poll, which prevents the agent from performing other work in the meantime. Furthermore, EgoSpaces primitives provide no mechanism for grouping operations transactionally. We introduce reactions to address the former concern and transactions to address the latter. We then combine the two constructs to build an even more powerful reactive construct.

Reactions. Like other mobile systems [13, 12], EgoSpaces provides reactive programming which allows agents to adapt to the presence of particular tuples. An EgoSpaces reaction associates a pattern with a set of actions to perform when a tuple matches the pattern. A reaction fires once for every matching tuple in the view. Disabling and re-enabling a reaction causes it to fire again for all matching tuples. Similarly, disconnection followed by reconnection causes reactions to re-fire. A reaction can remove its trigger from the tuple space and/or output the trigger modified in some way. This modification is achieved through a *tuple_modifiers* subroutine that can add, remove, or change fields in the tuple. For example, if an agent with unique id `ID1` retrieves the tuple:

```
⟨(ID, TupleID, 5), (dest, AgentID, ID1), (timestamp, time, 8:41), (temp, celsius, 28)⟩
```

and wants to change the time stamp, remove the destination, and add an owner, it defines the following *tuple_modifiers*:

```
tuple_modifiers(t) =
    {t.changeField(timestamp, currentTime), t.removeField(dest),
     t.addField(owner, AgentID, ID1) t.newID()},
```

The `newID` method allows the tuple's new owner to give it a new, unique id. If the tuple id generated was 12 and the time 9:36, the resulting tuple would be:

```
⟨(ID, TupleID, 12), (timestamp, time, 9:36), (temp, celsius, 28), (owner, AgentID, ID1)⟩
```

If the *tuple_modifiers* attempt to add a field that already exists, the current value of the field is replaced. The tuple output to the tuple space will have the same id (unless it is changed by the *tuple_modifiers*), and therefore the reactive construct will not fire repeatedly on the same tuple.

A reaction has one of two scheduling modalities, eager or lazy. Eager reactions occur immediately following the appearance of a matching tuple. Only other eager reactions can preempt them. A lazy modality brings a much weaker guarantee—the reaction eventually fires if the tuple remains in the view long enough. Other operations may occur in the meantime, possibly removing the tuple before the lazy reaction fires. Finally, reactions have a priority that arranges a hierarchy within each scheduling modality. Within each modality, reactions with higher priorities fire before reactions with lower priorities (the highest priority being 1). If more than one reaction with the same modality and same priority exists, the one fired first is chosen non-deterministically. If the first reaction removes the trigger, the second reaction will not fire. Reactions take the form:

$$\rho = \mathbf{react\ to}\ p\ [\mathbf{remove}]\ [\mathbf{and\ out}(tuple_modifiers(\tau))]$$

where the local name τ is bound to the trigger tuple; p is the reactive pattern; the optional keyword **remove** causes tuple removal; and the optional **out**(*tuple_modifiers*(τ)) outputs the trigger tuple with the *tuple_modifiers* applied. A reference agent enables and disables reactions using:

$$\begin{aligned} &\mathbf{enable}\ \rho\ \mathbf{with}\ sched_modality,\ priority\ \mathbf{over}\ \nu \\ &\mathbf{disable}\ \rho\ \mathbf{over}\ \nu \end{aligned}$$

where *sched_modality* is either eager or lazy, and *priority* is an integer. Reactions affect contributing agents' access controls; when specifying a view, the reference agent indicates if it intends to register reactions on it. Triggering the reaction and executing the associated actions occur as a single atomic step. If used, the **out** places a tuple in the reference agent's local tuple space at the completion of the reaction's execution. This tuple may trigger additional reactions.

Programming Example. Consider the application scenario in which the original sensors placed in the building generated Fahrenheit temperatures, but most sensors have been replaced by Celsius sensors. To provide a standard system, the Celsius sensors contain an agent that reacts to the presence of Fahrenheit readings, converts the values, and replaces the readings. Without the reactive construct, a programmer could use code similar to:

```

ν = [temperature data on this floor and adjacent ones]
p = ⟨⟨tempType, string, = "Fahrenheit"⟩⟩
while(true)
  sleep(time)
  data[] = ν.rdgp(p)
  if data ≠ null
    for i=1 to data.length
      ν.inp(data[i])
      data[i].changeField(tempType, "Celsius")
      data[i].changeField(tempValue, convert(oldT))
      out(data[i])

```

This code is slightly simplified because it refers to the Fahrenheit temperature as “oldT”, but this value must really be retrieved from the tuple (`data[i]`). The programmer must manage this code independent of the agent’s other operations. The agent creates and executes the thread to “enable” the reaction, and stops it to “disable” the reaction. In this example, the thread awakens periodically to check the reactive condition. The thread first reads all tuples matching p from the tuple space and executes the actions for the tuple.

With the built-in reactive construct, the code becomes:

```

ν = [temperature data on this floor and adjacent ones]
p = ⟨(tempType, string, =“Fahrenheit”)⟩
t_m(t) = {t.changeField(tempType, “Celsius”),
          t.changeField(tempValue, convert(oldT))}
ρ = react to p remove and out t_m(τ)
enable ρ with eager, 1 over ν

```

In this example, the programmer enables a high priority, eager reaction. From the programmer’s perspective, not only does this reactive construct simplify the code, it adds subtle, useful semantics. Instead of polling as in the first example, the reaction is guaranteed to fire immediately following the insertion of a matching tuple unless another eager reaction fires and removes the tuple. In the first example, tuples may be inserted and removed before the thread awakens to check for matches. Because the behavior is built into EgoSpaces, its actions can be optimized. Instead of gathering all possible matches each time before determining if the tuples have previously been processed, EgoSpaces can perform this check at each remote host. Finally, the application programmer has encapsulated the reaction and can reuse it on other views if desired.

Transactions. From an agent’s perspective, performing several operations sequentially is not atomic because other operations can interleave. For example, if an agent performs a successful **rdp** operation and then immediately attempts to **in** the same tuple, the **in** operation might be unsuccessful if another agent has, in the meantime, removed the tuple. At times, an application may want a sequence of operations to be atomic with respect to all other operations on the involved views. For example, if an application wants to replace a piece of data with an update, but does not want it to ever appear to the world that the data is unavailable, it needs to group the removal and replacement into a single atomic step. To accomplish this, we introduce *transactions* to EgoSpaces.

A transaction is a named sequence of actions that can include plain code, probing operations, and tuple creation. Because transactions must complete, they cannot include blocking operations that could halt the transaction indefinitely. Transactions are individual atomic actions; their intermediate results are not visible to the outside.

When creating a transaction, the reference agent provides a view restriction listing the involved views and serving as a contract between the reference agent and EgoSpaces. Any attempt inside the transaction to perform operations outside the view restriction generates an exception. The view restriction makes a

deadlock-free implementation of the transaction mechanism possible (see Section 5). A transaction takes the form:

$$T = \mathbf{transaction\ over}\ v_1, v_2, \dots \mathbf{begin}\ op_1, op_2, \dots \mathbf{end}$$

where T is the transaction's name; $v_1, v_2 \dots$ is the view restriction; and op_1, op_2, \dots is the sequence of operations. An agent executes a transaction using:

$$\mathbf{execute}\ T$$

Augmenting Reactions. In the previous reactive construct, the only actions an agent can perform are trigger removal and the output of an augmented version of the trigger tuple. We augment reactions by allowing them to execute a transaction in response to the appearance of a matching tuple. Because the transaction operates over the reference agents' views, it must execute from the reference agent itself. If the tuple triggering the reaction is local (i.e., in the reference agent's tuple space), the triggering of the reaction and the execution of the transaction can be grouped as a single atomic step. We consider this case first. If the tuple triggering the reaction is not local, it is not possible to trigger the reaction and execute the transaction in a single atomic step. We discuss this case second.

When the trigger tuple is local, we refer to the augmented reaction as an *extended reaction*, which has the form:

$$\rho = \mathbf{react\ to}\ p\ [\mathbf{remove}]\ [\mathbf{and\ out}(tuple_modifiers(\tau))]\ \mathbf{extended\ by}\ T(\tau)$$

An agent enables an extended reaction using:

$$\mathbf{enable}\ \rho\ \mathbf{with}\ sched_modality, priority\ \mathbf{over}\ \nu_l$$

Upon enabling, EgoSpaces verifies that ν_l is a local view, restricted in scope to only the reference agent.

In the second case, when the trigger tuple is not local, trigger, removal, and notification are a single atomic action, while the execution of the associated transaction is a separate action. The most important ramification of this subtle difference is that the trigger might not be available when the transaction executes because other operations can interleave with the reaction's triggering and the transaction. The transaction does, however, receive a copy (τ) of the trigger tuple. This type of reaction, a *followed reaction*, has the form:

$$\rho = \mathbf{react\ to}\ p\ [\mathbf{remove}]\ [\mathbf{and\ out}(tuple_modifiers(\tau))]\ \mathbf{followed\ by}\ T(\tau)$$

The use of the word **followed** in place of **extended** indicates the separation of the transaction's execution. The enabling mechanism for followed reactions is identical to extended reactions but not limited to a local view.

Programming Example. Imagine an agent that averages temperatures generated by sensors on the current floor over the past hour and replaces the old temperature readings with an average. To implement this behavior without reactions, a programmer writes something like:

```

ν = [Celsius temperature data on current floor]
p = ⟨⟨(timestamp, time, minutes = :00)⟩⟩
seenTuples = new Vector()
while(true)
  sleep(time)
  data = ν.rdp(p)
  if data ≠ null
    if !seenTuples.contains(data)
      p1 = ⟨⟨(temp Value, any, any), (timestamp, time, [within past hour])⟩⟩
      temps[] = ν.inpg(p1)
      avg = average(temps[])
      average = [tuple with average information]
      out(average)
      seenTuples.add(data)

```

With the built-in construct the code consists of defining a reaction:

```

ν = [Celsius temperature data on current floor]
p = ⟨⟨(timestamp, time, minutes = :00)⟩⟩
T(τ) = transaction over ν
  begin
    p1 = ⟨⟨(temp Value, any, any), (timestamp, time, [within past hour])⟩⟩
    temps[] = ν.inpg(p1)
    avg = average(temps[])
    average = [tuple with average information]
    out(average)
  end
ρ = react to p followed by T(τ)
enable ρ with eager, 1 over ν

```

The programmer explicitly declares the views over which its transaction will act. This contract allows the system to provide atomicity guarantees associated with the execution of the operations; the transaction executes as a single atomic step, while in the hand-coded case, each operation may interleave with other operations.

3.2 Behavioral Extensions

The reactive constructs make programming with EgoSpaces more flexible and provide more powerful semantics and guarantees. Most importantly, they allow agents to define general-purpose responses to trigger tuples in a view. In some cases, the types of actions an application performs will be common with other applications. This section classifies three such behaviors and expresses their semantics in terms of reactions. Building these behaviors into the system reduces the programming burden in common cases. In this section, we describe data migration, data duplication, and event capture. We also leave the system open to extension if additional coordination mechanisms arise in the future.

A reference agent attaches behaviors to views, and, as long as the behavior is enabled, encountering certain conditions triggers an automatic action. In general,

behaviors share several key components. First, a behavior responds to a trigger, identified via a pattern. Like reactions, behaviors respond once to each matching tuple. If tuples leave the view and return or the behavior is disabled and re-enabled, the behavior executes again.

Behaviors can be either eager or lazy. Eager behaviors execute as soon as the trigger is matched, and only other eager constructs can preempt them. Lazy behaviors eventually execute if the behavior remains enabled and the trigger stays present. Behaviors can also include tuple modifiers, which allow the reference agent to insert, change, or remove fields in resulting local tuples. How this is used will become apparent as we present the different behaviors. Finally, behaviors have an optional transaction executed at the behavior’s completion. In general, behaviors take the form:

$$\beta = \mathbf{act}(p) \ [tuple_modifiers(\tau)] \ [\mathbf{followed\ by} \ T(\tau)]$$

where **act** is the name of the behavior (e.g., “migrate” or “duplicate”). The operation list in a view specification includes behaviors, and contributing agents consider this set when evaluating access control functions. Reference agents enable and disable behaviors using:

enable β **with** *sched_modality* **over** ν
disable β **over** ν

We discuss each behavior individually, providing a brief description and syntax. We then show the behaviors’ semantics. For each behavior, we also include a programming example.

Data Migration. Mobile agents encounter a lot of data, but both data and agents constantly move. An agent may want to collect certain data without explicitly reading each piece. When the consistency of data is important, agents cannot make duplicates of data items and operate on them because other agents might operate on the originals. A common solution is replica management, where copies of data are kept consistent, but this solution is impractical in ad hoc environments where agents carrying originals and duplicates meet unpredictably. In transparent data migration, only one copy of the data item exists, and the migration behavior allows an agent to collect data matching a provided pattern. For example, building engineers might respond to work orders generated by distributed components. A single engineer should take responsibility for each work order because if multiple engineers pick up the same job, work will be wasted. When an engineer encounters a work order he should perform, the work order should move to the engineer.

When a migration is enabled, all matching tuples in the view automatically move to the reference agent. Because EgoSpaces evaluates contributing agents’ access control functions before determining which tuples belong to the view, contributing agents implicitly allow tuple transfer. Once migrated, the tuples become subject to the reference agent’s access controls, and this may affect the contents of other views. If desired, a migration uses tuple modifiers to change migrated tuples. For example, an engineer collecting work orders might mark the migrated tuples as “assigned” to prevent the work orders from migrating again.

Semantics. A migration reduces to a reaction that removes the trigger and generates a new tuple in the reference agent’s tuple space:

$$\begin{aligned} \mathcal{M} &= \text{migrate } p \text{ [tuple_modifiers}(\tau)\text{]} \\ &\triangleq \rho_m = \text{react to } p \text{ remove and out}(\text{tuple_modifiers}(\tau)) \end{aligned}$$

If the programmer supplies the optional tuple modifiers, the tuple placed in the local tuple space is the trigger tuple with the tuple modifiers applied. Otherwise, the tuple is exactly the trigger tuple. Even though the migrated tuple is the same tuple (unless the tuple modifiers change the ID), tuple migration may trigger reactions in the new location that have already fired for the tuple in the previous location. Enabling a migration reduces to enabling the reaction using the migration’s scheduling modality and a low priority (e.g., 10):

$$\begin{aligned} &\text{enable } \mathcal{M} \text{ with } \text{sched_modality} \text{ over } \nu \\ &\triangleq \text{enable } \rho_m \text{ with } \text{sched_modality}, 10 \text{ over } \nu_r \end{aligned}$$

where ν_r is ν with an added constraint that eliminates the reference agent. This prevents the EgoSpaces system from “migrating” tuples that are already local. The priority scheme maximizes the number of behaviors that execute, i.e., it ensures that duplicates are made before tuples migrate. A migration’s low priority allows other reactions and behaviors of the same modality to trigger first. If any of these actions remove the tuple, however, the migration will not occur.

Programming Example. The following code shows how a programmer would accomplish migration using only the basic EgoSpaces constructs. This code implements the work order collection application described above.

```

ν = [work orders on this floor and adjacent ones]
νr = [data in ν not owned by this agent]
p = ⟨⟨assigned, boolean, =false⟩⟩
while(true)
  sleep(time)
  data[] = νr.rdgp(p)
  if data ≠ null
    for i=1 to data.length
      ν.inp(data[i])
      data[i].changeField(assigned, true);
      out(data[i])

```

The tuple output has the same id as the one read, but the “assigned” field has been set to true. This implementation might miss matching tuples if they happen to appear and disappear while the thread is sleeping. To ensure local tuples are not infinitely migrated, the programmer must explicitly define ν_r , or the remote portion of a view ν . The definition of ν_r prevents tuples in the local tuple space (e.g., work orders created by this engineer that other engineers should perform) from being “migrated” to their current host.

Using the built-in migration behavior, the declaration of ν_r is hidden from the programmer.

```

ν = [work orders on this floor and adjacent ones]
p = ⟨(taken, boolean, =false)⟩
t_m(t) = {t.changeField(taken, true)}
M = migrate p t_m(t)
enable M with eager over ν

```

Because this behavior is integrated with the system, we can guarantee, for eager migrations, that tuples are migrated if they appear in the reference agent’s view, conditional on no other reactive constructs removing the tuple first.

Data Duplication. Under different circumstances, data availability is more important than data consistency, and an application would rather duplicate data items to make them available upon disconnection, with the knowledge that duplicates will not remain consistent with the originals. A duplication behavior copies tuples matching a pattern and places the copies in the reference agent’s local tuple space, leaving the originals unaffected. In our example application, the building engineer may collect sensor data for processing off-site. The engineer does not want to remove the data because others may need it.

Duplicated tuples may match the original view specification and be infinitely duplicated. They may also satisfy view specifications of other agents. As was the case with the migration behavior, applications deal with these concerns individually using tuple modifiers, e.g., by tagging all duplicates with a new field. Access to the copies becomes the responsibility of the owning agent. Again, because replica management proves too costly, duplicates do not remain consistent with originals, even if both persist in the view.

In some applications, an agent may respond to a particular tuple and generate an entirely new tuple in response. Data duplication can accomplish this by using the tuple modifiers to remove all of the fields and add all new fields.

Semantics. Duplication reduces to a reaction that does not remove the trigger and generates a new tuple in the reference agent’s tuple space. This new tuple must have a unique id.

```

D = duplicate p tuple_modifiers(τ)
  ≜ tuple_modifiers'(τ) = {τ.newID()}
  ρ_d = react to p and out(tuple_modifiers(τ) ∪ tuple_modifiers'(τ))

```

A duplication which specifies no tuple modifiers creates an exact copy (with a new tuple id), while one that adds a field “copied” marks all duplicates.

Enabling a duplication reduces to enabling the reaction with the provided scheduling modality and a high priority (e.g., 1):

```

enable D with sched_modality over ν
  ≜ enable ρ_d with sched_modality, 1 over ν

```

A high priority ensures duplication occurs before other actions, e.g., migration.

Programming Example. Using only the EgoSpaces primitive operations, an engineer duplicating structural integrity data he encounters on the current floor and the adjacent floors would use code similar to:

```

ν = [structural agent data on this floor and adjacent ones]
p = ⟨⟨(strain, number, any), (acoustic emission, number, any),...⟩
seenTuples = new Vector()
while(true)
  sleep(time)
  data[] = ν.rdg(p)
  if data ≠ null
    for i=1 to data.length
      data[i].newID()
      out(data[i])
      seenTuples.add(data[i])

```

The `seenTuples` data structure prevents the agent from duplicating the same data multiple times.

Using the built-in duplication behavior reduces to defining a view, creating a duplication behavior, and enabling it on the view:

```

ν = [structural agent data on this floor and adjacent ones]
p = ⟨⟨(strain, number, any), (acoustic emission, number, any),...⟩
D = duplicate p
enable D with eager over ν

```

This eager behavior is guaranteed to duplicate all matching tuples that appear in the view without missing any, while the hand-coded example may miss some. A lazy duplication has semantics identical to those of the hand-coded example.

Event Capture. The `EgoSpaces` primitives, reactions, and behaviors operate over the state of the system by interacting with data. Many applications also benefit from reacting to events raised in the system. For example, an agent might want to be notified when another agent accesses a piece of data. In our system, examples of events include the arrival of a new view contributor and another agent's data access operations.

`EgoSpaces` events are special tuples. An agent registers its interest in an event via a pattern over such tuples. Once registered, event notifications for events matching the pattern propagate to the reference agent. To prevent superfluous event generation, `EgoSpaces` raises event tuples only for specific registrations, and the event's callback execution consumes the event tuple created for it. This allows multiple registrations for the same event such that when a matching event occurs, all registered parties receive notification. A reference agent uses a transaction to specify the event's callback.

Semantics. The event behavior reduces to a pair of reactions. The first generates a copy of the event tuple augmented with the id of an event registration and places it in the reference agent's local tuple space. The second reacts to the generated tuple and executes the callback:

$$\begin{aligned}
\mathcal{E} &= \text{event}(p) \text{ followed by } T_e(\tau) \\
&\triangleq \text{eid} = \text{new event id} \\
&\quad \rho_{e1} = \text{react to } p \text{ and out}(\tau \oplus \{\langle \text{eID}, \text{event id}, \text{eid} \rangle\}) \\
&\quad \rho_{e2} = \text{react to } (p \oplus \{\langle \text{eID}, \text{event id}, = \text{eid} \rangle\}) \text{ remove extended by } T_e(\tau)
\end{aligned}$$

The \oplus indicates that the provided field, in this case the new event id, is added to the tuple. The generation of the event copy and the callback execution are not an atomic action. However, the reference agent can prevent other agents from stealing its event tuples using its access control function.

Enabling an event behavior reduces to enabling the two reactions:

$$\begin{aligned}
&\text{enable } \mathcal{E} \text{ with } \text{sched_modality} \text{ over } \nu \\
&\triangleq \text{enable } \rho_{e1} \text{ with } \text{eager}, 1 \text{ over } \nu \\
&\quad \text{enable } \rho_{e2} \text{ with } \text{sched_modality}, 1 \text{ over } \nu_l
\end{aligned}$$

The first reaction (generating a personal copy of the event) has **eager** modality and high priority, guaranteeing the reference agent is notified. The second reaction's scheduling modality corresponds to the behavior's modality and also executes at high priority. This reaction is enabled on a local view (ν_l) defined specifically for this behavior that contains only local event tuples.

This behavior's semantics differ slightly from the others. Every event behavior, **eager** or **lazy**, is guaranteed to be triggered because an event tuple is created specifically for each registration. In the **lazy** case, however, by the time the callback executes, the entity that caused the event may be no longer connected.

This reduction assumes mechanisms exist to generate events and clean up event tuples. The former is discussed in Section 4, and the latter is accomplished by a reaction that removes event tuples:

$$\rho_{gc} = \text{react to } p \text{ remove}$$

where p matches any event tuple. This is an **eager** reaction with a priority of at least 2, guaranteeing all event copies have been generated (at priority 1):

$$\text{enable } \rho_{gc} \text{ with } \text{eager}, 2 \text{ over } \nu_e$$

This reaction is defined and enabled on every agent's event view, so an agent need not define it each time it enables an event behavior.

Programming Example. Because event capture requires an event generation mechanism, there is no way to accomplish this same behavior using the initial EgoSpaces operations. Assume that a tuple indicating the arrival of a new host is represented with an event tuple similar to the following:

$$\langle \langle \text{eventType}, \text{string}, \text{hostArrival} \rangle, (ID, \text{HostID}, \text{newHost}), \dots \rangle$$

If the building engineer wants to receive notification of the arrival of an inspector on adjacent floor, his application agent has the following code:

```

ν = [this floor and adjacent ones]
p = ⟨(eventType, string, =hostArrival)⟩
Te(τ) = transaction over null
      begin
        [display message to user]
      end
E = event(p) followed by Te(τ)
enable E with eager over ν

```

The null view restriction indicates that the transaction does not use any views.

4 Design Strategies

The extensions presented build on the EgoSpaces middleware. In some cases (e.g., event generation, reaction registration), the new features are integrated into the core system, while others build on top of the system.

View Construction and Maintenance. View construction and maintenance protocols directly influence the operations’ implementations. Inefficient view building limits performance. Our initial efforts have led to the development of a network abstractions protocol that, given neighborhood restrictions requested by the reference agent, provides a list of qualifying agents, represented as a tree. For details of this protocol, see [16]. In short, the protocol builds the tree and maintains it in the face of mobility.

Basic Operations. An efficient implementation of blocking operations takes advantage of reactions to prevent expensive polling. For example, an **ing** operation entails a (low priority) eager reaction that does not remove its trigger. When this reaction fires, a transaction follows and attempts an **inpg**. If this operation returns anything other than ϵ , the operation returns and disables its associated reaction. If the operation is unsuccessful, another operation removed the tuple first. Because these operations are serializable with respect to the view, this is within the operation’s semantics.

Atomic probes are transactions performed on a single view. They require locking all view participants, performing the operation, and unlocking the participants. This locking mechanism is discussed below in the description of the transaction implementation. Agents benefit from intelligent view definition, as this type of operation becomes costly on views involving large numbers of agents.

A variety of possible implementations for scattered probes exist. The simplest implementation polls the view’s participants in order (by id). When a match is found, it is removed if the operation is an **in** and returned. If all participants have been queried and no match found, the operation returns ϵ . Group operations query all participants and return all matches. More sophisticated implementations of the single operations can take advantage of the environment; for example, one might query the physically closest agents first.

Transactions. A transaction must operate over several views with explicit guarantees that its internal state is not visible from outside. As such, transactions are inherently costly. EgoSpaces reduces this cost by requiring a reference agent to explicitly declare what other agents need to be blocked for the duration of the transaction by providing a list of views. Because the agents contributing to each view are known, EgoSpaces can lock the transaction’s participants (including the reference agent) in order (by id). If any other agent also performs a transaction, it locks agents in the same order, avoiding deadlock. If a contributing agent moves out of the view while a transaction is locking agents, it must be unlocked before departing. If the transaction’s operations are already executing, the agent’s departure must be delayed until the transaction completes. We guarantee enough time to complete the transaction before the agent disappears from communication range using *safe distance* [18]. If a new agent moves into the view while a transaction is in progress, its arrival is delayed until the transaction completes.

Reactions. Because the reactive mechanism lies at the core of the EgoSpaces extensions, an efficient implementation is essential. Each agent keeps a reaction registry (containing all reactions it has registered) and a reaction list (containing all reactions this agent should fire on behalf of other agents, including itself). A reaction registry entry contains a reaction’s id, the tuple to output when the reaction fires (if any), and the transaction that extends or follows this reaction (if any). A reaction list entry contains the reaction’s id, the reaction issuer’s id, the reaction’s pattern, the view’s data pattern, and a boolean indicating whether or not to remove the trigger. Upon registration, the reaction propagates to all view participants and is inserted in each participant’s reaction list. For all matching tuples in the view, the reaction fires. This firing sends a notification (containing a copy of the trigger) to the registering agent. If specified, the tuple is removed from the tuple space. While the reaction remains enabled, new tuples in the view are checked against the pattern. For each match, the registering agent receives a notification and locates the reaction in the reaction registry. If necessary, it performs the appropriate **out** operation and schedules any associated transaction.

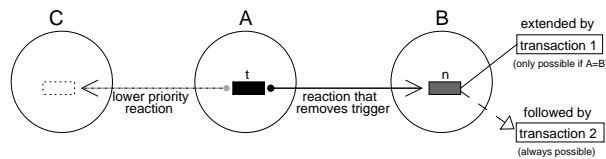


Fig. 2. The Reaction Mechanism

In Fig. 2, agents B and C register reactions, which both match t . The reaction with the highest priority (B’s reaction) fires first, generating notification n for B. Because this reaction removes the trigger, C’s lower priority reaction will not fire. B’s reaction can be extended or followed by a transaction. The former is only allowed when the trigger is local (i.e., $A=B$).

During the view’s construction, new agents receive the reaction registration and add it to their reaction list. As new agents move into the view’s scope, they receive any registered reactions. As agents move out of the view, they remove

information regarding registered reactions. If these agents return, they receive the registrations and fire the associated reactions again for matching tuples.

Behaviors. Because the semantics of behaviors are written as reactions, their implementation relies on the reaction’s implementation. Again, the key reason for building these behaviors into the system is to provide common actions as simple operations and to allow for code encapsulation and reuse.

Event Generation. To successfully implement event capture, we add an event raising mechanism to EgoSpaces. Some example event types include host arrival and departure, agent arrival and departure, and data access operations. Each type of operation has a defined type string (e.g., *hostArrival*) and some secondary information (e.g., the *HostID* for a host arrival or departure event). The event generation mechanism raises an event only if an agent has registered for the event. Upon generation, special event tuples are created for each registered agent, and these tuples are transmitted to the agent. The event’s callback then executes according to the registration’s modality (eager or lazy).

5 Conclusion

The success of a coordination middleware for ad hoc mobile environments lies in its ability to address the key issues of this constrained environment. First, the amount of information available necessitates mechanisms to easily and abstractly limit one’s operating context. Second, the middleware must provide programming abstractions tailored to specific application domains while remaining general enough to maintain a small footprint on devices with constrained memory requirements. Finally, the communication restrictions and responsiveness requirements inherent in wireless applications direct design. The original EgoSpaces model began to address the first of these three concerns. The additional constructs and behavioral extensions introduced in this paper complete this task and provide the needed high-level coordination mechanisms. The reduction of the behaviors into a unifying construct, the reaction, decreases the required middleware support. With such a direct attack on complexities specific to ad hoc mobile networks, EgoSpaces and its extensions promise to transform application development in this environment. Additionally, this paper shows how these behavioral extensions serve as a powerful abstraction for practical systems.

ACKNOWLEDGEMENTS

This research was supported in part by the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. Broch, J., Johnson, D.B., Maltz, D.A.: The dynamic source routing protocol for mobile ad hoc networks. Internet Draft (1998) IETF Mobile Ad Hoc Networking Working Group.
2. Ko, Y., Vaidya, N.: Location-aided routing (LAR) in mobile ad hoc networks. In: Proc. of MobiCom. (1998) 66–75
3. Park., V., Corson, M.S.: Temporally-ordered routing algorithm (TORA) version 1: functional specification. Internet Draft (1998) IETF Mobile Ad Hoc Networking Working Group.
4. Royer, E., Toh, C.K.: A review of current routing protocols for ad hoc mobile wireless networks. IEEE Personal Communications (1999) 46–55
5. Harter, A., Hopper, A.: A distributed location system for the active office. IEEE Networks **8** (1994) 62–70
6. Want, R., et al.: An overview of the PARCTab ubiquitous computing environment. IEEE Personal Communications **2** (1995) 28–33
7. Ryan, N., Pascoe, J., Morse, D.: Fieldnote: A handheld information system for the field. In: 1st International Workshop on TeloGeoProcessing. (1999)
8. Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: A mobile context-aware tour guide. ACM Wireless Networks **3** (1997) 421–433
9. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proceedings of MobiCom, ACM Press (2000) 20–31
10. Salber, D., Dey, A., Abowd, G.: The Context Toolkit: Aiding the development of context-enabled applications. In: Proc. of CHI'99. (1999) 434–441
11. Hong, J., Landay, J.: An infrastructure approach to context-aware computing. Human Computer Interaction **16** (2001)
12. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4** (2000) 26–35
13. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proc. of the 21st Int'l. Conf. on Dist. Comp. Systems. (2001) 524–533
14. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proc. of the 10th Int'l. Symp. on the Foundations of Software Engineering. (2002) 21–30
15. Picco, G.P., Murphy, A.L., Roman, G.C.: On global virtual data structures. In Marinescu, D., Lee, C., eds.: Process Coordination and Ubiquitous Computing. (2002) 11–29
16. Roman, G.C., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proc. of the 24th Int'l. Conf. on Software Engineering. (2002) 363–373
17. Gelernter, D.: Generative communication in Linda. ACM Trans. on Prog. Lang. and Systems **7** (1985) 80–112
18. Roman, G.C., Huang, Q., Hazemi, A.: Consistent group membership in ad hoc networks. In: Proc. of the 23rd Int'l. Conf. on Software Engineering. (2001)